Develop Models with Hibernate

23

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- How to wire hibernate to access database.
- How to perform CRUD operation on database with DAO.
- Basic yet important methods such as save and saveOrUpdate.

23.1 | Installing MySQL

We first need to create a database in which we can create the tables we need. For this example, we are using the MySQL database so you need to install it on your local computer. If you have not installed and setup MySQL, you can use the local server environment that will help you, and install and manage it with ease. Depending on your operating system, you could use one of the versions of MySQL explained in Sections 23.1.1 and 23.1.2. Please go through their documentation to learn about how to use them to set up MySQL.

23.1.1 Windows

WampServer: WampServer is the most useful local server environment for Windows users. It is free and opensource. It provides installation and management for Apache2, PHP, and MySQL.

```
Download: http://www.wampserver.com/en/
Documentation: http://www.wampserver.com/en/
```

HeidiSQL: This is a simple-to-use MySQL client for windows which allows you to manage your database easily. It can connect to local as well as remote servers. HeidiSQL offers a lot of features and can connect to many database systems such as MariaDB, MySQL, Microsoft SQL, and PostgreSQL.

```
Download: https://www.heidisql.com/download.php
Documentation: https://www.heidisql.com/help.php
```

23.1.2 Windows/Mac

MAMP: This is the Mac counterpart of the WampServer. Although different companies developed them, they have similar features in terms of managing local servers. MAMP is free to use. MAMP also has a Windows version so you could use this if you like this interface better than WampServer.

```
Download: https://www.mamp.info/en/downloads/
Documentation: https://documentation.mamp.info/
```

Once you setup MySQL on your local machine, if you are not comfortable using command line, you may use one of the following free graphical user interface (GUI) applications to create database and tables.

Sequel pro: This is a free to use MySQL client. It offers a simple to use interface and good features. Although it does not offer the most advanced features, it is alright for small projects.

```
Download: https://sequelpro.com/download
Documentation: https://sequelpro.com/docs
```

23.2 | Create Database and Tables

After setting up MySQL and your desired GUI, it is time to create our database. For our eShop example, we will give the same name to our database as our application. So create a database using the GUI and give it a name "MyEShop" as shown in Figure 23.1.

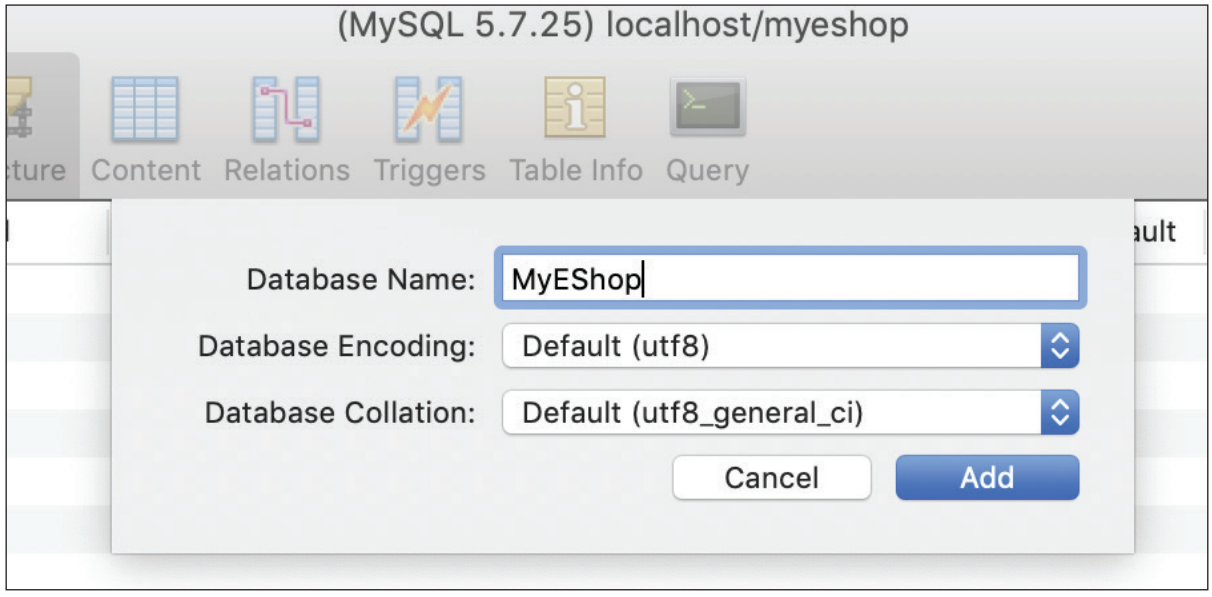


Figure 23.1 Adding a new database in GUI.

Once you click on the Add button, GUI will create the database for you. Now, it is time to add a table. You can either use the GUI to add it or use the following SQL which you can run in the Query window to create the Customer table with the fields we need.

```
CREATE TABLE 'customer' (
    'customerid' int(11) unsigned NOT NULL AUTO_INCREMENT,
    'name' varchar(50) DEFAULT NULL,
    'email' varchar(50) DEFAULT NULL,
    'address' varchar(100) DEFAULT NULL,
    PRIMARY KEY ('customerid')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

See the following screenshot in Figure 23.2 to see how to run this query in the GUI.

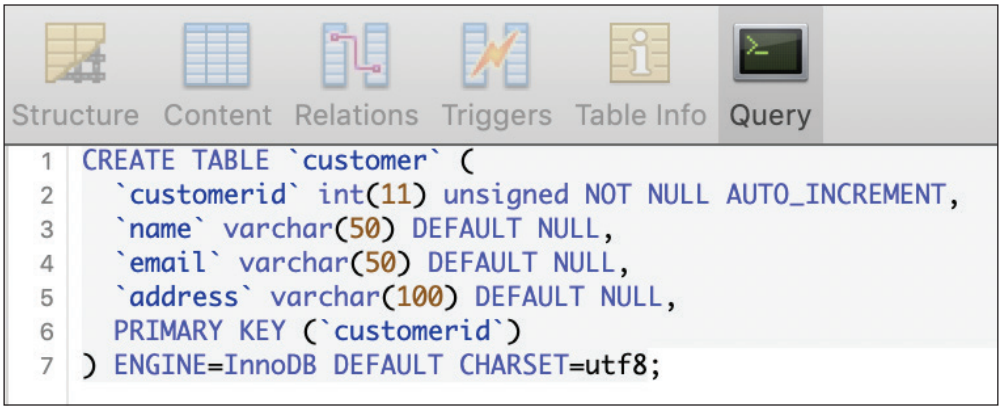


Figure 23.2 Query window to run CREATE TABLE command.

Please note that every table should have id column so we can identify the record and link it to other tables as foreign key. In this case, we will be more specific and call our id column as "customerid". This is the field we have not added in our Customer model yet so we need to update our model to accommodate this field.

23.2.1 Linking Tables to Models

We will add customerid field as shown in the code in the following image, and let Spring know that this is an id by adding @Id annotation from javax.persistence package.

Since this is an id column, we would like it to be incremental. This is what we need to tell Hibernate as well. For this, we will be using @GeneratedValue annotation from javax.persistence package.

Customer.java

```
1  package com.fullstackdevelopment.myeshop.model;
2
3  import java.sql.Timestamp;
4  import javax.persistence.Entity;
5  import javax.persistence.GeneratedValue;
6  import javax.persistence.Id;
7  import javax.persistence.Table;
8
9  @Entity
10 @Table(name="customer")
11 public class Customer {
12
13     @Id
14     @GeneratedValue
15     private Long customerid;
16     private String name;
17     private String address;
18     private Timestamp dateOfBirth;
19
20     public Long getCustomerid() {
21         return customerid;
22     }
23     public void setCustomerid(Long customerid) {
24         this.customerid = customerid;
25     }
```

Flash Quiz How do the values in customerid table get stored if @GeneratedValue annotation is removed?

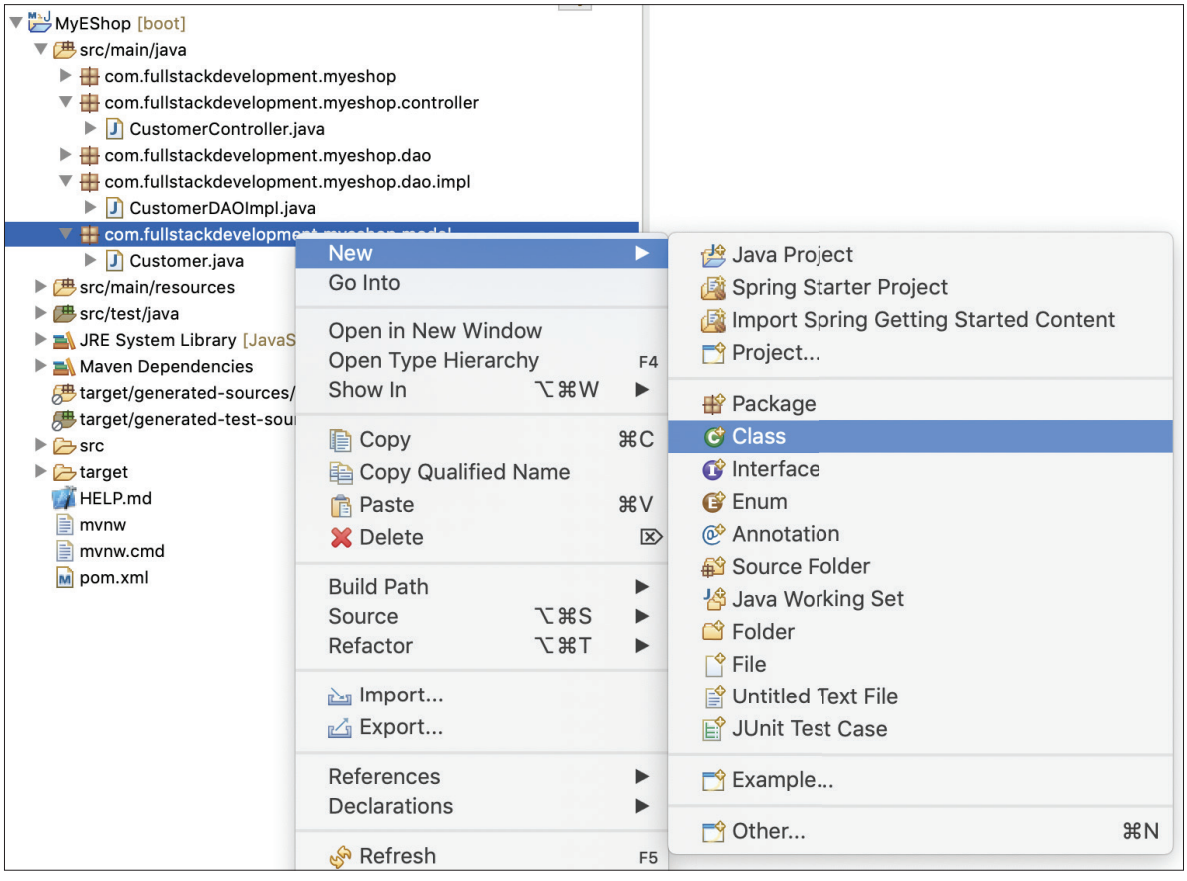


Figure 23.3 Menu to create a new class.

There is one more important step to map this model to the customer table we have created. This step we have already done in Chapter 22 by adding @Table annotation to our Customer class name. We also specified the table name that is mapped to this model by using name attribute @Table(name="customer").

In Chapter 22 we talked about making address as a separate entity so we can store multiple addresses for the customer. So, let us work on that now. We need to update our Customer table and model for this. Let us first create a new model called Address, with basic address related fields. Right-click on the "com.fullstackdevelopment.myeshop.model" package and click on New->Class.

After clicking on this menu, you will see a window to add details about the model. Let us call this class as "Address" as shown in Figure 23.4.

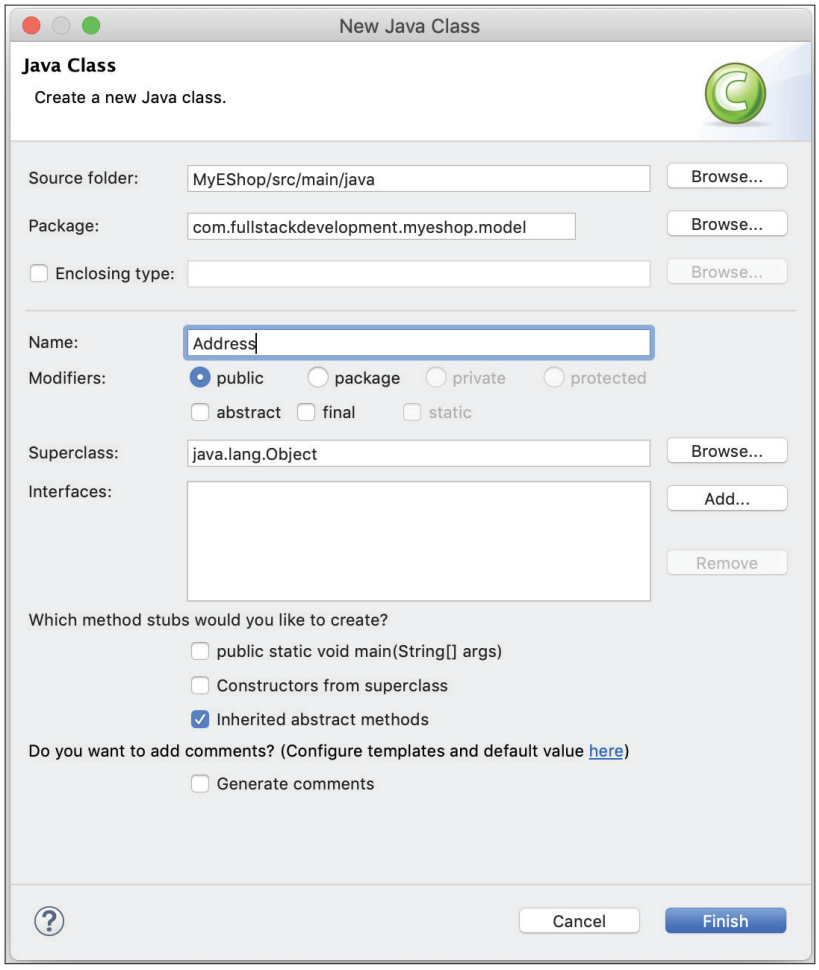


Figure 23.4 New class detail window.

Upon clicking on the Finish button, it will create the Address class under "com.fullstackdevelopment.myeshop.model" package as shown in Figure 23.5.

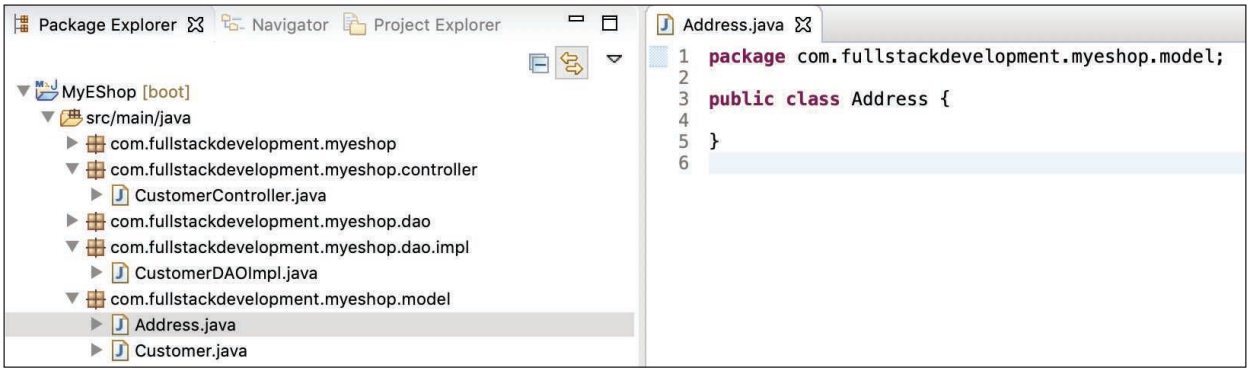


Figure 23.5 Newly added Address class.

Once the class is created, add basic address related fields and "addressid" as an id field with the required annotations.

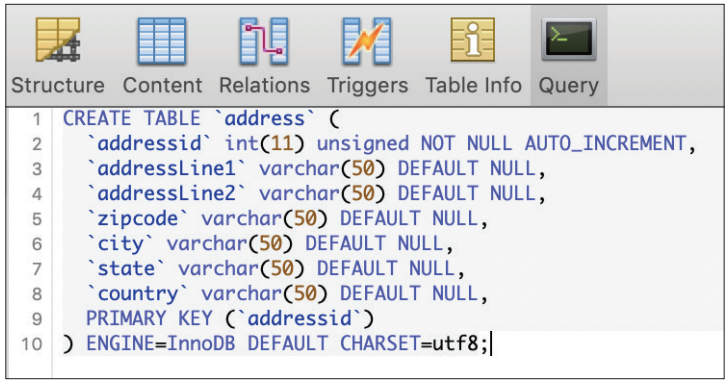
Address.java | Customer.java

```
1  package com.fullstackdevelopment.myeshop.model;
2
3  import javax.persistence.Entity;
4  import javax.persistence.GeneratedValue;
5  import javax.persistence.Id;
6  import javax.persistence.Table;
7
8  @Entity
9  @Table(name="address")
10 public class Address {
11
12     @Id
13     @GeneratedValue
14     private Long addressid;
15     private String addressLine1;
16     private String addressLine2;
17     private String zipcode;
18     private String city;
19     private String state;
20     private String country;
21
22     public Long getAddressid() {
23         return addressid;
24     }
25     public void setAddressid(Long addressid) {
26         this.addressid = addressid;
27     }
28     public String getAddressLine1() {
29         return addressLine1;
30     }
31     public void setAddressLine1(String addressLine1) {
32         this.addressLine1 = addressLine1;
33     }
34     public String getAddressLine2() {
35         return addressLine2;
36     }
37     public void setAddressLine2(String addressLine2) {
38         this.addressLine2 = addressLine2;
39     }
40     public String getZipcode() {
41         return zipcode;
42     }
43     public void setZipcode(String zipcode) {
44         this.zipcode = zipcode;
45     }
46     public String getCity() {
47         return city;
48     }
49     public void setCity(String city) {
50         this.city = city;
51     }
52     public String getState() {
53         return state;
54     }
55     public void setState(String state) {
56         this.state = state;
57     }
58     public String getCountry() {
59         return country;
60     }
61     public void setCountry(String country) {
62         this.country = country;
63     }
64 }
65
```

As you can see, we have added the basic fields and @Entity annotation along with @Table annotation and specified our table as "address". Now, let us create this table in the database as well. You can run the following SQL in the Query window of the GUI.

```
CREATE TABLE 'address' (
    'addressid' int(11) unsigned NOT NULL AUTO_INCREMENT,
    'addressLine1' varchar(50) DEFAULT NULL,
    'addressLine2' varchar(50) DEFAULT NULL,
    'zipcode' varchar(50) DEFAULT NULL,
    'city' varchar(50) DEFAULT NULL,
    'state' varchar(50) DEFAULT NULL,
    'country' varchar(50) DEFAULT NULL,
    PRIMARY KEY ('addressid')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

See the following image to see how to run this query.



QUICK CHALLENGE Based on the entity relationship diagram, create SQL for other tables.

23.2.2 Setting up Relationships

Now, we need to link this table with Customer so Customer can store multiple addresses. For this, we need to update our SQL as well as our Model code. Let us first update our table to link address to customer. We can use "customerid" as a foreign key in the address table.

First add "customerid" as a field in the "address" table. You can use the following command for that.

```
ALTER TABLE 'address' ADD COLUMN customerid int(11) not null;
```

Please note that we are not yet adding a database constraint as foreign key on the address table. We are just creating customerid field in the address table so hibernate can link these two tables for us.

Now, we need to update our Address class to add this newly added field. However, we will not add this field as Long but we will reference it to the Customer object. Later, we will see how to link these tables in the model code.

```
8  @Entity
9  @Table(name="address")
10 public class Address {
11
12     @Id
13     @GeneratedValue
14     private Long addressid;
15     private String addressLine1;
16     private String addressLine2;
17     private String zipcode;
18     private String city;
19     private String state;
20     private String country;
21     private Customer customer;
22
```

You will also need to add Getter and Setter for this field.

```
65      public Customer getCustomer() {
66          return customer;
67      }
68      public void setCustomer(Customer customer) {
69          this.customer = customer;
70      }
```

Now, it is time to let Spring know how we need to access Customer and Address. As discussed earlier, every customer can have multiple addresses such as Shipping, Billing, etc. Hence, the customer can have one-to-many relationship with address. Similarly, address can have many-to-one relationship with customer. This is what we need to define in the code so Spring will know how to treat these two entities.

Before we specify in both the models, we first need to add address field in the Customer model. Since we would like to define one-to-many relationship, we will add address as List so we can get multiple addresses for one customer entry. See the code in the following image.

```
11  @Entity
12  @Table(name="customer")
13  public class Customer {
14
15      @Id
16      @GeneratedValue
17      private Long customerid;
18      private String name;
19      private String address;
20      private Timestamp dateOfBirth;
21      private List<Address> addresses;
22
23      public List<Address> getAddresses() {
24          return addresses;
25      }
26      public void setAddresses(List<Address> addresses) {
27          this.addresses = addresses;
28      }
```

Once we define this, we now need to define these relationships. For this, we can use @OneToMany and @ManyToOne annotations. See the Customer model code in the following image.

```
 1  package com.fullstackdevelopment.myeshop.model;
 2
 3  import java.sql.Timestamp;
 4  import java.util.List;
 5
 6  import javax.persistence.CascadeType;
 7  import javax.persistence.Entity;
 8  import javax.persistence.FetchType;
 9  import javax.persistence.GeneratedValue;
10  import javax.persistence.Id;
11  import javax.persistence.OneToMany;
12  import javax.persistence.Table;
13
14  @Entity
15  @Table(name="customer")
16  public class Customer {
17
18      @Id
19      @GeneratedValue
20      private Long customerid;
21      private String name;
22      private String address;
23      private Timestamp dateOfBirth;
24
25      @OneToMany(mappedBy="customer",fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
26      private List<Address> addresses;
27
```

@OneToMany annotation has mappedBy property where we can define the variable reference from the Address class. Earlier, we have added field **private** Customer customer; in the address class. This is what we need to reference here.

Flash Quiz What will happen if we remove @OneToMany annotation from the addresses field?

Then we have another property called fetch. This is an important property to set as it defines FetchType. In our case, we have used LAZY as the fetch type. It means Hibernate will not query tables until we ask for it specifically. This will speed up the fetch operation as Hibernate does not need to define join while running select query on the Customer table. If we explicitly ask to give customer addresses then Hibernate will add the join query. This helps in increasing application performance. The other fetch type is called EAGER. This fetch type instructs Hibernate to get addresses each time we ask for a customer object. It adds a little penalty on the performance side.

The last property that you see is "cascade". This property defines what happens when a CRUD operation is performed on customer object. There are many cascade types you can use, as follows.

1. **CascadeType.PERSIST:** When used, `save()` or `persist()` operations cascade to related entities.
2. **CascadeType.MERGE:** When used on the owning entity, all the related entities of the owning entity are merged upon the owning entity is merged.
3. **CascadeType.REFRESH:** When used, related entities are refreshed.
4. **CascadeType.REMOVE:** When used, upon deletion of the owning entity, all the related entities get removed.
5. **CascadeType.DETACH:** When used, in case of "manual detach" all the related entities are detached.
6. **CascadeType.ALL:** When used, it performs all the above cascade operations.

In order to get it working properly, we also need to tell Hibernate about Address to Customer relationship. Hence, we need to define @ManyToOne relationship on the customer object in the Address class.

Address.java | Customer.java

```
1  package com.fullstackdevelopment.myeshop.model;
2
3  import javax.persistence.CascadeType;
4  import javax.persistence.Entity;
5  import javax.persistence.FetchType;
6  import javax.persistence.GeneratedValue;
7  import javax.persistence.Id;
8  import javax.persistence.JoinColumn;
9  import javax.persistence.ManyToOne;
10 import javax.persistence.Table;
11 import com.fasterxml.jackson.annotation.JsonBackReference;
12
13 @Entity
14 @Table(name="address")
15 public class Address {
16
17     @Id
18     @GeneratedValue
19     private Long addressid;
20     private String addressLine1;
21     private String addressLine2;
22     private String zipcode;
23     private String city;
24     private String state;
25     private String country;
26
27     @JsonBackReference
28     @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
29     @JoinColumn(name="customerid", referencedColumnName = "customerid")
30     private Customer customer;
31
```

As you can see, we have added the following three lines on the Customer object declaration.

```
@JsonBackReference
@ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
@JoinColumn(name="customerid", referencedColumnName = "customerid")
```

This will tell Hibernate how to link address with customer.

We have already learned fetch and cascade, so let us jump on to the other two annotations. @JoinColumn annotation tells Hibernate about the column in the database that we need use for the link. This is the foreign key field we have created in the address table. The name property allows us to specify this field name from the table, in our case it is "customerid". The referencedColumnName property allows us to specify the column name (primary key) of the customer class, which is linked with address via "customerid" field. There is one more annotation @JsonBackReference you can see on top of @ManyToOne annotation. This annotation is designed to solve the infinite recursion problem by linking two-way – one for Parent and other for Child.

Now, let us look at another type of relationship. When two entities are related to each other in a singular fashion, it is known as one-to-one relationship. This type of relationship is defined by @OneToOne annotation. The owning class field will have this annotation.

QUICK CHALLENGE State the differences of @OneToOne, @OneToMany, and @ManyToOne.

23.3 | Making DAO to Perform CRUD

We can now move on to the DAO class section to write code that can perform CRUD operations on these models.

In our CustomerDAOImpl class, we will be adding this interface code, which will hide the model details and only expose it via CustomerDAO interface methods and hence further hides the implantation from the calling classes. Since we will be performing CRUD operations, we need to make sure the transactions are handled properly. For this, we can simply annotate the class with @Transactional annotation. This tells Spring that we need Spring's help to manage the transactions for this class.

CustomerDAOImpl.java

```
1  package com.fullstackdevelopment.myeshop.dao.impl;
2
3  import java.util.Collection;
4  import java.util.Optional;
5
6  import javax.transaction.Transactional;
7
8  import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
9  import com.fullstackdevelopment.myeshop.model.Customer;
10
11 @Transactional
12 public class CustomerDAOImpl implements CustomerDAO{
13
```

If we asked Spring to manage transactions on our behalf by annotating the class as @Transactional, Spring will create a proxy class which is invisible at runtime and provides a way to inject behavior into the object before, after, or around method calls.

Now, we can move on to write code for findCustomerByEmail(String email) method. In order to use Hibernate to perform CRUD operations, we need SessionFactory from the Hibernate package. As shown in the code in the following image, we will use @Autowired annotation to inject SessionFactory to CustomerDAOImpl class.

CustomerDAOImpl.java

```
1  package com.fullstackdevelopment.myeshop.dao.impl;
2
3  import java.util.List;
4
5  import org.hibernate.SessionFactory;
6  import org.springframework.beans.factory.annotation.Autowired;
7
8  import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
9  import com.fullstackdevelopment.myeshop.model.Customer;
10
11 public class CustomerDAOImpl implements CustomerDAO{
12
13     @Autowired
14     private SessionFactory sessionFactory;
15
```

We also need to create a constructor and pass SessionFactory parameter, which will be configured in the XML configuration.

```
public CustomerDAOImpl(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}
```

Once we get the sessionFactory object, we can add the following code to fetch a Customer record by email. Before we proceed with this task, we need to add the email field in the Customer model class. We have already added this into the database, so it is time to add in the model.

Customer.java

```
1  package com.fullstackdevelopment.myeshop.model;
2
3  import java.sql.Timestamp;
13
14 @Entity
15 @Table(name="customer")
16 public class Customer {
17
18     @Id
19     @GeneratedValue
20     private Long customerid;
21     private String name;
22     private String email;
23     private Timestamp dateOfBirth;
```

As you can see from the code in the above image, we have removed the String type "address" field and added String type "email" field. We have also created getter and setter for this field. Let us proceed with adding customer search code with email address.

```
51     @Override
52     public Customer findCustomerByEmail(String email) {
53         CriteriaBuilder criteriaBuilder = sessionFactory.getCurrentSession().getCriteriaBuilder();
54         CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
55         Root<Customer> root = criteriaQuery.from(Customer.class);
56         criteriaQuery.select(root).where(criteriaBuilder.equal(root.get("email"), email));
57         Query<Customer> q = sessionFactory.getCurrentSession().createQuery(criteriaQuery);
58         return q.getSingleResult();
59     }
60
```

In the above example, we have used CriteriaBuilder from "javax.persistence.criteria.CriteriaBuilder" package. It is accessible via sessionFactory. This CriteriaBuilder object is then used to get CriteriaQuery instance. Then the "from" method is used to set the Customer class as "root" on this query object. Now the root is set, we used "where" method to set the comparison on email column. Then we used create query object for this CriteriaQuery that we built. And finally, we run "`q.getSingleResult()`" to get the desired record for the email parameter we are passing to this method.

QUICK CHALLENGE With help of findCustomerByEmail(String email) example, write code for get(int id) and getAll methods.

Let us move on to write code for the save method. In Hibernate, there are three options we can use to save the record in the database.

1. **`save():`** As the name suggests, this is used to save an entity into the database. It also returns a generated identifier. If the entity already exists in the database, it throws an exception.
2. **`persist():`** It is very similar to save method. The only difference is it does not return a generated identifier but returns a void.
3. **`saveOrUpdate():`** This is useful when you do not know if the entity exists in the database. This method will either save or update the entity into the database. Hence, it does not throw an exception as `save()` method throws if the entity already exists in the database; it will simply update it.

The code in the following image shows how to save record through Hibernate and how easy it is to set data without writing a single line of SQL.

```
@Override
public Long save(Customer t) {
    Customer customer = t;

    if (customer == null) {
        customer = new Customer();
        customer.setName("Albert Einstein");
        customer.setEmail("Albert@Einstein.com");

        // set date of birth
        SimpleDateFormat format = new SimpleDateFormat("yyyyMMdd");
        try {
            Date parsed = format.parse("1879014");
            Timestamp dateOfBirth = new Timestamp(parsed.getTime());
            customer.setDateOfBirth(dateOfBirth);
        } catch (ParseException e) {
            System.out.println("Error in parsing the date of birth");
            e.printStackTrace();
        }

        // set billing and shipping addresses
        List<Address> allAddresses = new ArrayList<Address>();

        Address shippingAddress = new Address();
        shippingAddress.setAddressLine1("112");
        shippingAddress.setAddressLine2("Mercer St");
        shippingAddress.setCity("Princeton");
        shippingAddress.setZipcode("NJ 08540");
        shippingAddress.setCountry("USA");
        sessionFactory.getCurrentSession().save(shippingAddress);
        allAddresses.add(shippingAddress);

        Address billingAddress = new Address();
        billingAddress.setAddressLine1("112");
        billingAddress.setAddressLine2("Mercer St");
        billingAddress.setCity("Princeton");
        billingAddress.setZipcode("NJ 08540");
        billingAddress.setCountry("USA");
        sessionFactory.getCurrentSession().save(billingAddress);
        allAddresses.add(billingAddress);

        customer.setAddresses(allAddresses);
    }

    sessionFactory.getCurrentSession().save(customer);
    sessionFactory.getCurrentSession().flush();

    return customer.getCustomerid();
}
```

In the above code, multiple steps are taking place. We are first checking if the parameter is null or not. If it is, say, null, we are initializing a Customer object. Now, we will use this customer object to set field values. If it is not null then we are directly saving it using the sessionFactory. In the null case, setting up name and email are straightforward processes. For setting up date of birth, we need to convert the given date into a Timestamp with the help of SimpleDateFormat. After that, we are declaring an ArrayList for Address entity. This ArrayList will hold all the addresses. We are creating a shippingAddress by initializing an Address object and setting up the field data. After that we are saving the address entity into the database with the help of sessionFactory. We are repeating the process for the billingAddress. Once the addresses are saved, we are adding this list to the customer object's addresses field and later saving it into the database. In the end, we are flushing the database so database is updated. Since `save()` method returns generatedId (in our case customerid), we are simply returning it.

QUICK CHALLENGE By using save method example shown in Section 23.3, write code for update method.

Now, let us write code for "delete" method. There are two types of entities we will be dealing with when deleting from the database. One is transient and another one is persistence. Since the transient entity is not associated with the session, we use identifier to remove the instance from the database. For example, if Customer object is transient then we can delete it by specifying ID of it. See the following code block.

```
Customer customer = new Customer();
customer.setCustomerid(1L);
sessionFactory.getCurrentSession().delete(customer);
```

The above code will delete a Customer record which contains customerid = 1.

In case of persistence, we load the entity first and then delete it. We use sessionFactory object to load the entity.

```
Customer customer = sessionFactory.getCurrentSession().load(Customer.class, 1L);
if(customer != null) {
sessionFactory.getCurrentSession().delete(customer);
}
```

In the above example, we are first loading Customer using sessionFactory and then making sure it is not null before executing delete operation on it.

In our case, since we are getting a customer object via parameter, we do not need to load this entity. We will just make sure that the entity is not null before running the delete operation on it.

```
@Override
public void delete(Customer t) {
    Customer customer = t;
    if(customer != null) {
        sessionFactory.getCurrentSession().delete(customer);
    }
}
```

Summary

In this chapter, we have learned how to install and use MySQL, create database and tables, and run queries using GUI. We have also seen how to use Hibernate to perform CRUD operations on a database. We have learned various relationships like @OneToOne and @OneToMany. We have also learned about performing CRUD operations on a database including the difference between save, persist, and saveOrUpdate.

Multiple-Choice Questions

1. What is RDBMS?
   (a) Relational Database Management System
   (b) Rational Data Batch Management System
   (c) Relational Database Messaging System
   (d) Rotational Database Management System
2. Which of the following is not a valid cascade type?
   (a) CascadeType.PERSIST
   (b) CascadeType.MERGE
   (c) CascadeType.SAVE
   (d) CascadeType.REFRESH
3. `save()` is used to update a recode in Hibernate.
   (a) True
   (b) False
4. SessionFactory is a thread safe object
   (a) True
   (b) False
5. _____ annotation is used to make value of id column incremental.
   (a) @GeneratedValue
   (b) @Id(Increment)
   (c) @Id(ColumnValue = "Increment")
   (d) @Incremental

Review Questions

1. What is MySQL?
2. What is the use of Database?
3. What is the difference between DBMS and RDBMS?
4. How to make sure that the class is treated as model to map with a table in the database?
5. What is Hibernate?
6. What is object-relational mapping?
7. What are the disadvantages of using object-relational mapping?
8. What is the difference between `save()` and `saveOrUpdate()`?
9. When can you use `persist()`?
10. What is CriteriaBuilder?

Exercises

1. Define all the different Database encodings and their effect on table.
2. Define all the different Database collations and their effect on table.
3. Change the column name "name" to "fullname" of the table customer we created in section 23.2.
4. Explain the use of @JsonBackReference and the effect on the table if removed from the field.

Project Idea

Create a simple application to create a TODO list application. Use Hibernate to store and retrieve data. Define all the relationships and make sure you use @OneToMany and @OneToOne relations.

Recommended Readings

1. Thorben Janssen and Steve Ebersole. 2017. *Hibernate Tips: More Than 70 Solutions to Common Hibernate Problems*. Createspace Independent Pub
2. K. Santosh Kumar. 2017. *Spring and Hibernate*. McGraw Hill: New York
3. Hibernate User Guide – https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html
4. Hibernate ORM Guide – https://hibernate.org/orm/documentation/5.4/